LAB 5 REPORT

Course: CPS633

Section: 6

Group Members:

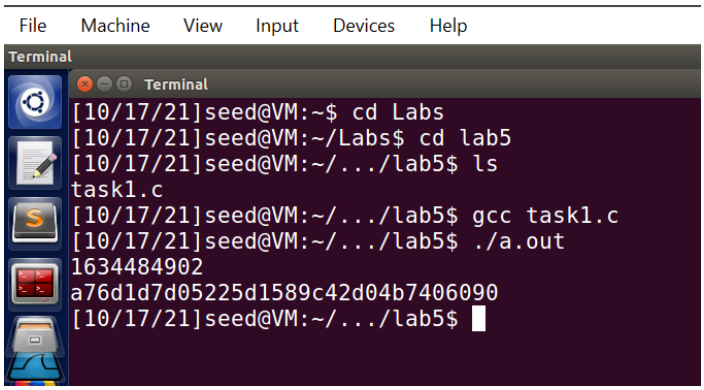
Alishba Aamir, 500974648

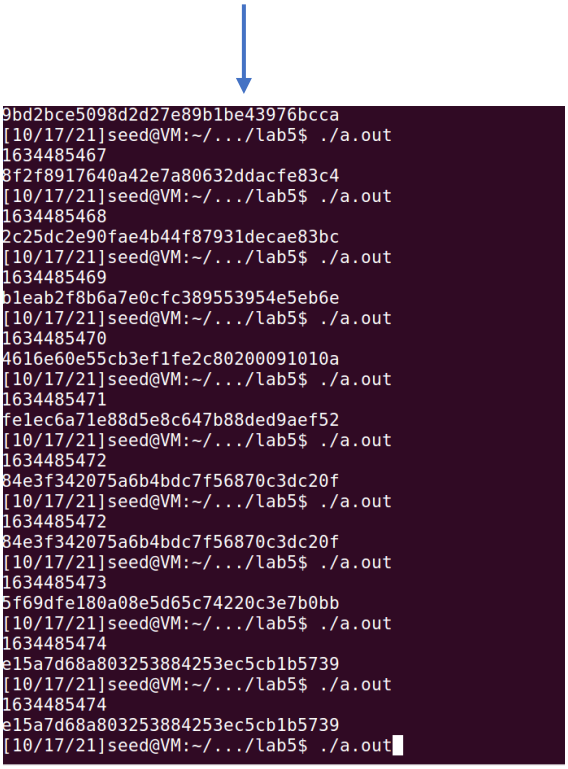
Ivan Golovine, 500813431

Fangbo Ren, 500884730

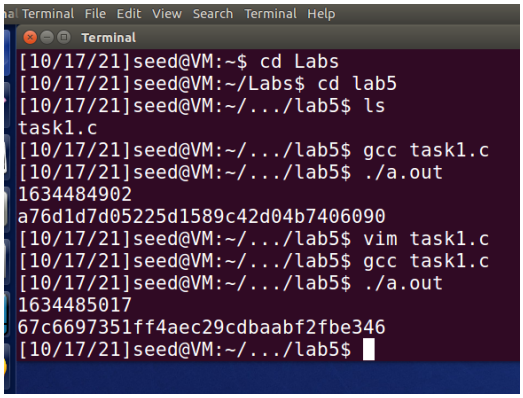
Task 1: Generate Encryption Key in a Wrong Way

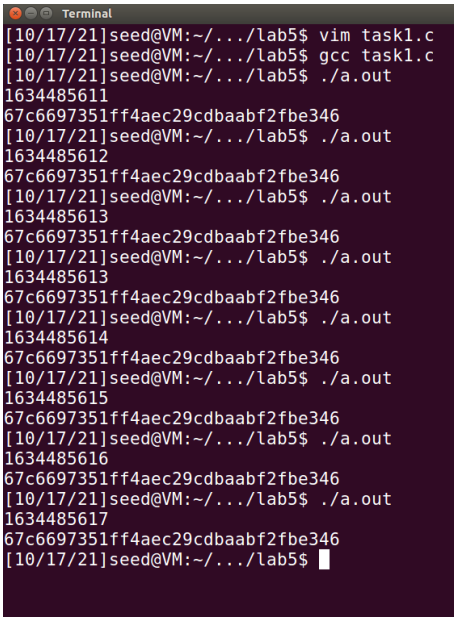
With srand (time(NULL));





Without srand (time(NULL));



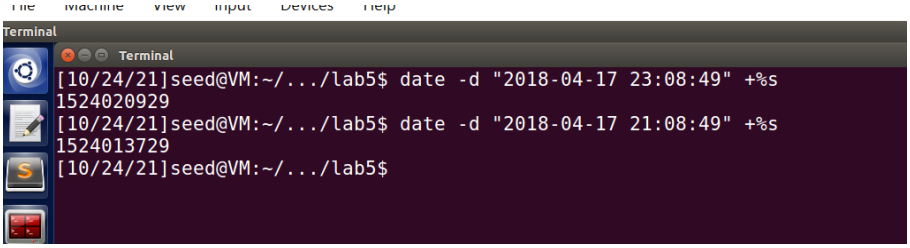


The purpose of the time() function is to return the current calendar time. If you run the prog **with** the srand (time(NULL)) a couple of times, you will notice the first line is the current time while the second line is the encrypted key. The srand(time(NULL)) function uses the computer clock(current calendar time) as the seed. Since time is always changing, the seed will also be changing. This means every time, you run the program, the encrypted key will be different. You can see the evidence by comparing the 2 outputs above. For the program running **without** the srand(time(NULL)) function, the time is different **BUT** the encrypted key is the SAME in all the runs. For the program running **with** the srand(time(NULL)) function, the encrypted key is the DIFFERENT With every run.

TASK 2: Guessing the Key

First, we are going to use the date command to find time in second for range of 2 hours for the timestamp that we were given. Therefore, the file could have been encrypted anywhere between

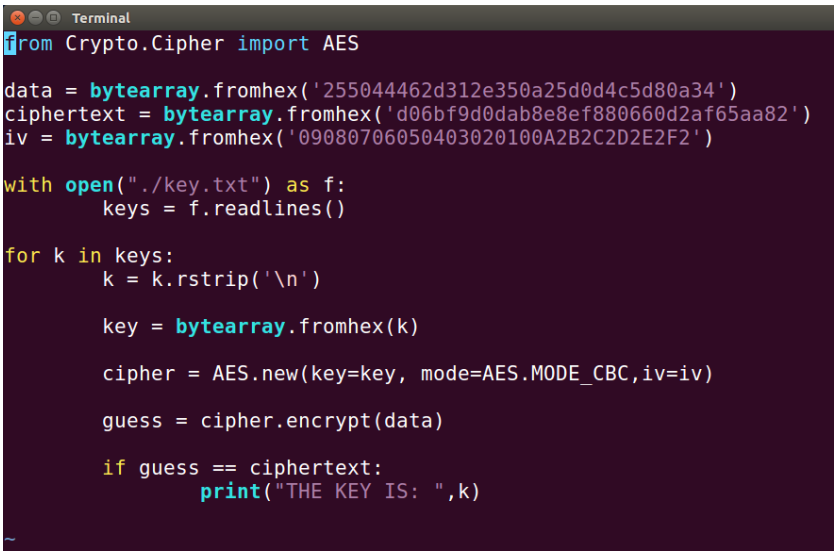
1524013729 to 1524020929.



Now, we used the c program given to us in Task 1 to generate the list of possible keys that could have been used to encrypt the document. Here, instead of using the current time, we incremented the time one by one starting from the first possible time, the key could have been made to the last possible time the key could have been made. We run this program and put the list of possible keys in key.txt.

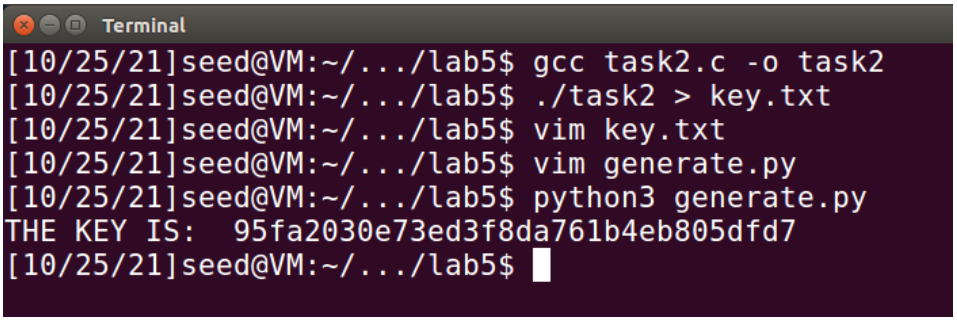
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
        int i;long long t;


        for (t=1524013729; t < 1524020929; t++){
                char key[KEYSIZE];
                srand (t);
                for (i = 0; i< KEYSIZE; i++){
                        key[i] = rand()%256;
                        printf("%.2x", (unsigned char)key[i]);

                }
                printf("\n");
}

}
```

Next, with the help of a python program, we took each key from the list of keys and used it along with the iv, and data to figure out the guess key. If our key and cipher text match, then that means we have successfully guessed the correct key used to encrypt the document.



As seen below, we run our task2.c prog to generate the list of possible keys. Then we run our generate.py program which uses the list of keys to guess a key.

Our guess for the key is : 95fa2030e73ed3f8da761b4eb805dfd7



Task 3: Measure the Entropy of Kernel

Initial Entropy

```
[10/24/21]seed@VM:~/.../lab5$ cat /proc/sys/kernel/random/entropy_avail
394
[10/24/21]seed@VM:~/.../lab5$
```

Moving the mouse would generally produce a slow consistent amount of entropy, when doing it in addition to other things it was effective but on its own it was descent for consistent production.

```
Terminal
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail   Sun Oct 24 17:56:48 2021

883
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail   Sun Oct 24 22:12:18 2021

893
```

Typing would be faster at building up the entropy but if there isn't a large amount of entropy built up would have periods of time where it would also jump lower.

Reading a large file would usually result in a big jump in entropy of 100 or more up or down, especially when combined with typing and mouse movement.

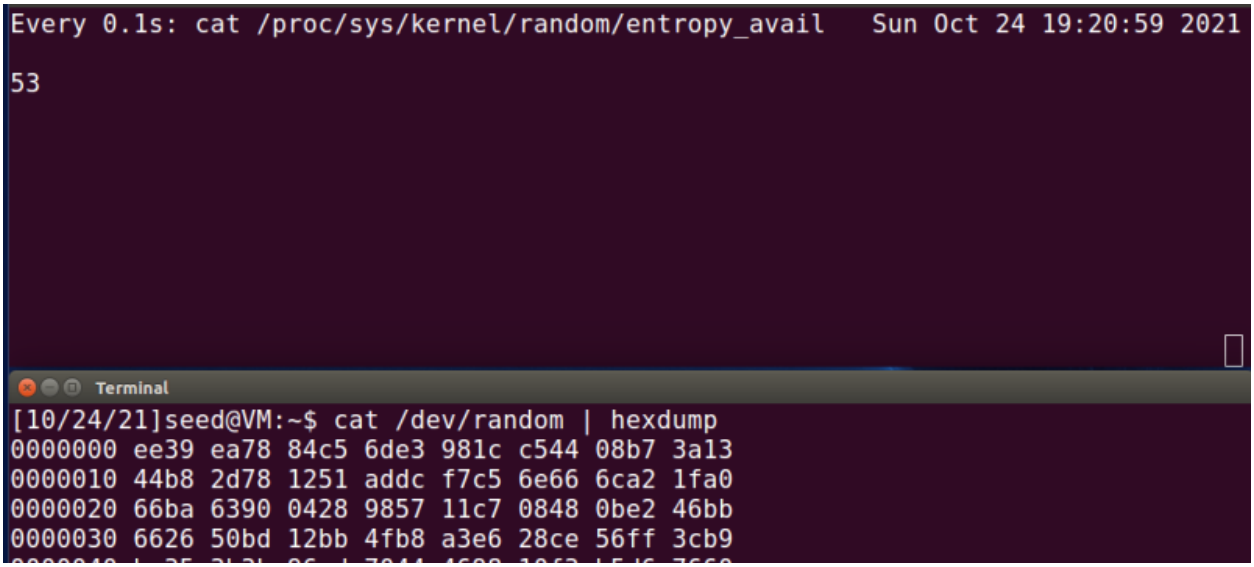
```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail   Sun Oct 24 18:13:11 2021

3327
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail   Sun Oct 24 18:17:19 2021

3549
```

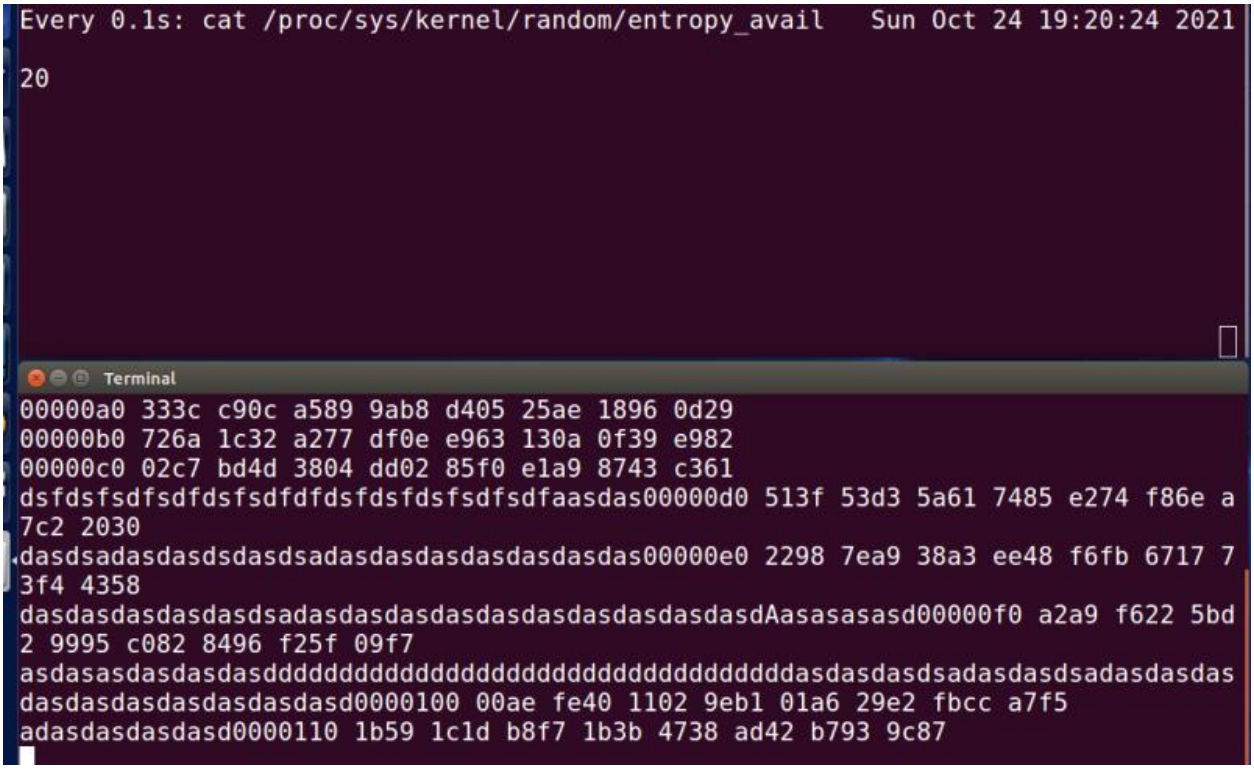
Visiting a website would result in similar jump up or down to opening a large file.

Task 4: Get Pseudo Random Numbers from /dev/random

When you don't move the mouse the number increases by 1 every second and then dumps in hexcode when it reaches the 60's in randomness.



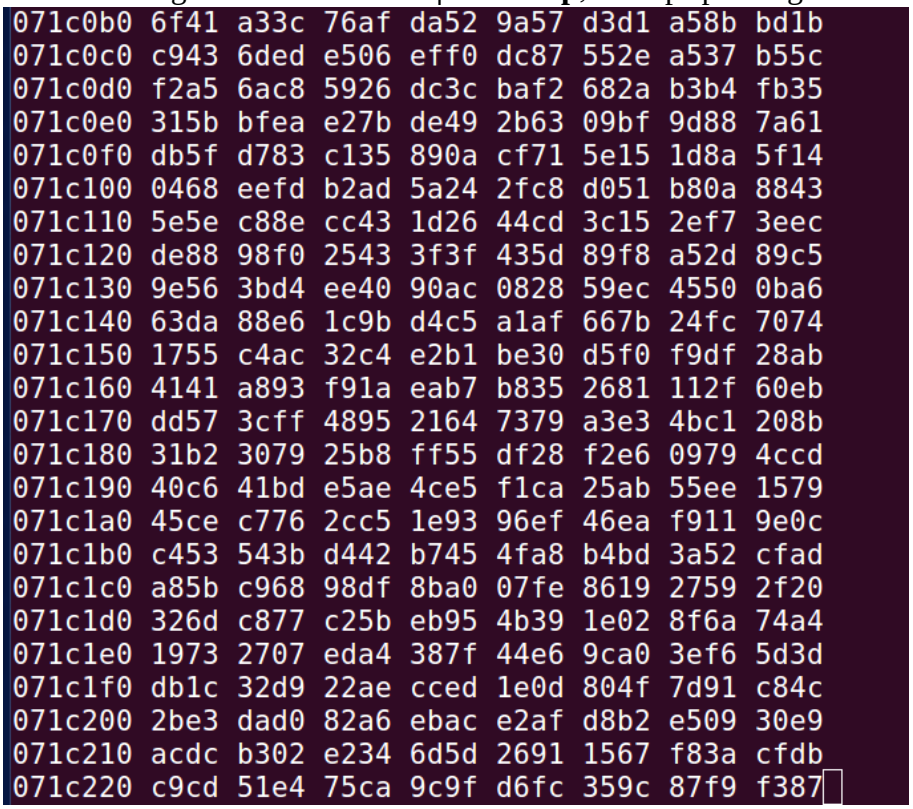
When you move the mouse or type it increases the random generator rapidly and the dump happens quicker. By inputting a lot of random movements with the mouse or keyboard it is possible to make it skip a hexdump when reaching the threshold.



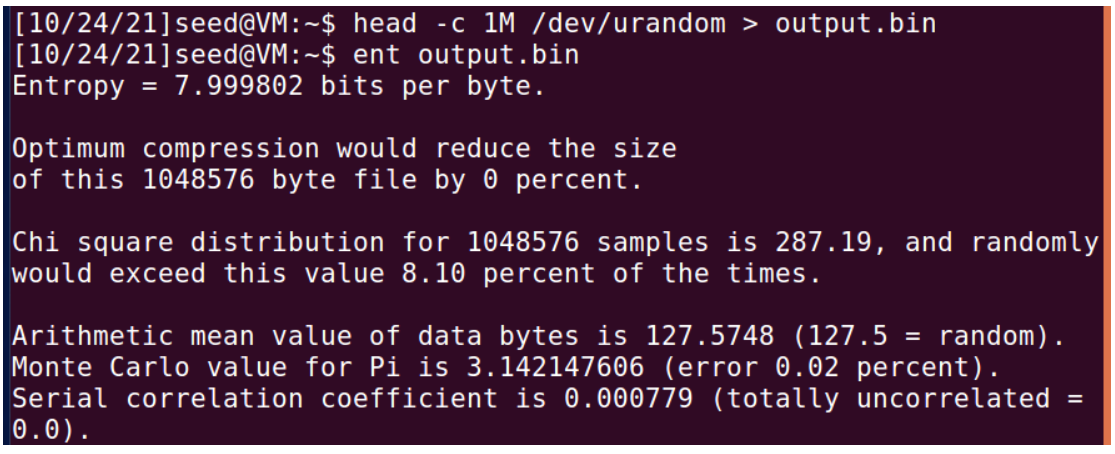
If there is not enough entropy available on the server, the server may result in blocking when creating another session key. By flooding it is possible to exhaust the kernels entropy pool and pause any further reads until a sufficient entropy is replenished.

Task 5: Get Random Numbers from /dev/urandom

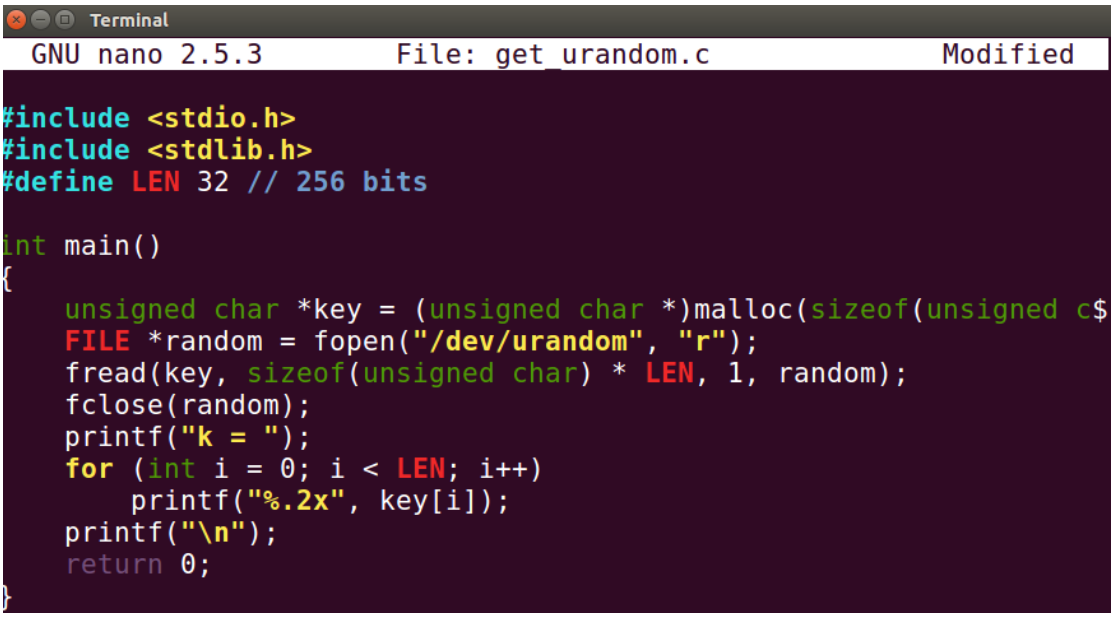
When running **cat/dev/urandom | hexdump**, it keeps printing out the random numbers.



The bellow image shows I truncate the first 1 MB outputs into a file named output.bin and Then use **ent** to evaluate its information density.



I modify the code, compile the get_urandom.c and run it to generate a 256-bit random number.




```
[10/24/21]seed@VM:~$ nano get_urandom.c
[10/24/21]seed@VM:~$ gcc get_urandom.c -o get_urandom
[10/24/21]seed@VM:~$ ./get_urandom
k = 385a7fc635bcb5bf0b06d0e882dcbd4d89c3bc8ddd64a64cc556a5557701383
e
[10/24/21]seed@VM:~$
```